SCOC: High-Radix Switches Made of Bufferless Clos Networks

Nikolaos Chrysos†, Cyriel Minkenberg†, Mark Rudquist*, Claude Basso*, and Brian Vanderpool*

†*IBM Research – Zurich, Switzerland*

**IBM Systems & Technology Group, Rochester, USA*

***Abstract*—In today's datacenters handling big data and for exascale computers of tomorrow, there is a pressing need for high-radix switches to economically and efficiently unify the computing and storage resources that are dispersed across multiple racks. In this paper, we present *SCOC*, a switch architecture suitable for economical IC implementation that can efficiently replace crossbars for high-radix switch nodes. SCOC is a multi-stage bufferless network with O($\frac{N^2}{m}$) cost, where m is a design parameter, practically ranging between $4-16$. We identify and resolve more than five fairness violations that are pertinent to hierarchical scheduling. Effectively, from a performance perspective, SCOC is *indistinguishable* from efficient flat crossbars. Computer simulations show that it competes well or even outperforms flat crossbars and hierarchical switches. We report data from our ASIC implementation at 32 nm of a SCOC 136×136 switch, with shallow buffers, connecting 25 Gb/s links. In this first incarnation, SCOC is used at the spines of a server-rack, fat-tree network. Internally, it runs at 9.9 Tb/s, thus offering a speedup of $1.45\times$, and provides a fall-through latency of just 61 ns.**

1. Introduction

Omnipresent since the early years of data communication and parallel computing systems, crossbars have become virtually indispensable in modern off-chip and on-chip networks. They are non-blocking and deterministically fair, thus offering essential performance guarantees to system designers. However, crossbars designs have two blocks that are hard to scale: A datapath core, with $2N$, long, multi-bit links and N^2 crosspoints; and a scheduler block, with order O(N^2) point-to-point wires, and several, random-logic, arbitration units. Typically, EDA tools can handle up to a fixed number of gates per hard block or tile [17]. Having passed a threshold, the placement and routing effort grows exponentially, which can delay the design.

At the same time, in exascale supercomputers, but also in warehouse-scale datacenters, there is a pressing need for high-radix switches. The latter are advantageous building blocks, because they lower the cost and the power consumption (fewer chips, backplane wires, and cables). Additionally, they lower the packet delay, and, because they enable networks with fewer stages, they can provide more predictable performance in communication-intensive workloads [15].

Three years back, our team decided to scale a server-rack interconnect with a second-tier of high-radix spine switches [7]. When we tried to place and route a 32-port crossbar, our EDA tools did not face difficulties [27], but the 136-port crossbar that we were targeting was much tougher to build. We first considered to apply bit slicing and manual placement, but, eventually, we resorted to a hierarchical design for the 136-port crossbar, which our tools could handle seamlessly.

In this paper, we present *Scalable Clos On-Chip (SCOC)*, a switch node architecture that can be used as an efficient crossbar replacement. SCOC is a combined-input-output-queued (CIOQ) switch [8] with *virtual output queues (VOQs)*, built around a *bufferless*, non-blocking, Clos network [9], with cost that grows as O($\frac{N^2}{m}$). Integer m is a design parameter, practically in the range of $4-16$.

Whereas *buffered* fabrics are preferred for off-chip networks [5], in this paper we show through the example of SCOC that the situation is reversed inside the chip. The lack of internal buffers allows SCOC to use *packet-level multipathing*, which delivers consistent performance irrespective of the spatial orientation in the workload. The catch, of course, is that a bufferless Clos needs global scheduling [4], which, in practice, performs sub-optimally. SCOC takes advantage of the abundance of wires that are available on-chip to remedy its scheduling inefficiencies using *cheap on-chip speedup*. The combination of the two, packet-level multipathing and speedup, empowers SCOC with remarkable performance.

The scheduling subunit of SCOC is also hierarchically organized. Using a gradual contention-resolution scheme, SCOC reduces the number of requests per arbiter, and also trims down the inter-arbiter connections, thus lowering the wiring congestion inside the scheduling tile. SCOC features a *time-sliced pipeline* that computes port-level pairings and route-assignments simultaneously. This scheduling pipeline is deep enough to amortize the time complexity over multiple clock cycles.

Absorbingly, from a performance perspective, SCOC is *indistinguishable* from efficient flat crossbars. Maintaining this service level proved to be an uphill, but worthwhile struggle. The gradual contention resolution and the asynchronous operation of SCOC could seriously endanger the determinism expected of correct operation. We drafted a set of microbenchmarks that helped us identify more than five violations of fairness, and came up with practical solutions to resolve each one of them.

We have implemented a 136-port SCOC switch, for 25 Gb/s ports, and $m = 4$. This number of ports was selected in order to limit the periphery of the chip, which is dictated by the I/O cores. The chip is a 32 nm ASIC designed using standard EDA tools, operates at 454 MHz and consumes 150 Watt. Internally, the SCOC chip runs at approximately 9.9 Tb/s, thus over-provisioning the user bandwidth by a factor $s = 1.45$, and provides a fall-through packet latency of just 61 ns. In retrospect, being able to synthesize the arbiters and

Nikolaos Chrysos is currently with Foundation For Research and Technology (FORTH); email: nchrysos@ics.forth.gr

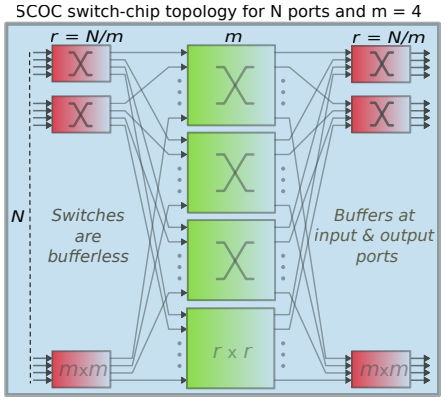


Fig. 1 SCOC (unidirectional Clos) topology for $m = 4$.

crossbars in an automated way enabled us to flexibly delve into the design process, add enhancements, and examine different system-level alternatives. If a semi- or full-custom approach had been used instead, any changes or additions would have incurred a significantly higher design cost.

A. Contributions & contents

In this paper, we make the following key contributions:

1) We present SCOC, a bufferless Clos network, suitable for integrated circuit implementation, that can readily replace crossbars in high-radix switches and networks on-chip.
2) We identify and resolve the unfairness violations that are pertinent to SCOC hierarchical scheduling, thus also providing a valuable set of tools for similar designs.
3) We evaluate the route-allocation scheme of SCOC, showing that, for large packets, it automatically converges to a maximal solution. Additional results show that SCOC is especially powerful under non-uniform traffic patterns, outperforming both hierarchical [15], [30] and flat crossbars.
4) We present the implementation of a 136×136 switch chip for 25 Gb/s links.

The remainder of this paper is organized as follows. In Sec. 2, we present the topology of SCOC and its hierarchical scheduler. Our fairness enhancement mechanisms are described in Sec. 3. In Sec. 4, we first study the route-allocation scheme of SCOC and present how it iteratively converges to maximal solutions. Then, we use computer simulations to evaluate our proposal and to compare it with other alternatives. Section 5 outlines the implementation of our switch chip. Finally, we discuss our design choices and related work in Sec. 6, and conclude in Sec 7.

2. SCOC Architecture

In this section we present the SCOC architecture. First, we outline the topology of SCOC and its packet buffers. Second, we describe the scheduler and its operation.

A. Network topology and packet buffering

SCOC is an on-chip network, with N input and N output ports, as shown in Fig. 1. Its topology is an (unfolded) three-stage Clos, with m middle-stage switches, and r input (or output) switches (or groups). Packet memory buffers are implemented at input and output ports. Besides these memories, the network is *bufferless*. We also considered to place buffers inside the middle-stage crossbars: We rejected this alternative to avoid its many small buffers as well as the blocking and the out-of-order packet delivery that these introduce.

In arbitrary Clos networks, the number of ports per group ("n") can differ from m, but we consider them to be equal. Thus, our network is *rearrangeably non-blocking*. The number of crosspoints is minimized when $m = \sqrt{N}$. However, taking into account high-level chip-partitioning and scheduling considerations, practical configurations of SCOC will likely use $m \ll r$.

A *route* is the end-to-end path of a packet from an input to an output port that passes through a middle-stage switch—there are no shorter paths. When referring to a particular input or output group, we use the term *route* to also denote the point-to-point, unidirectional link that connects it to a middle-stage switch. There are m possible routes for each packet, and m routes available to each input and output group.

Any packet from an input buffer is eligible for service. The packets that belong to the same connection are serviced in FIFO order, but are routed independently. Nevertheless, they always arrive at output buffers in order, because no queuing takes place inside the interconnect.

In our implementation, $N = 136$, $m = 4$, $r = 34$ and $s = 1.45$. For this design point, it was feasible to implement the (4) middle-stage crossbars as a single, four-times-wider (4-way) TDM crossbar, instead of four separate ones. Thus, we chose this alternative because our EDA tools could afford it. In this implementation, each input buffer can feed an input of the wide crossbar at full speed. This enables the *multiple-transfers-per-input* method, discussed in Sec. 3. For larger configurations ($m > 4$), it may be difficult to fit all middle-stage crossbars into a single tile and to have this large input-buffer speedup. However, as discussed in Sec. 4, the performance of SCOC with $m > 4$ does not depend heavily on this optimization.

B. The scheduler of SCOC

Scheduling in SCOC is realized by per-group and per-port arbiters that run *asynchronously* from one another, supporting variable-size transfers. There are three main scheduling functions: request, grant and accept, which complete in one clock cycle each, and operate in a pipeline fashion.

The scheduling block, depicted for $m = 4$ in Fig. 2, is partitioned into r *request*, r *grant* and r *accept units*. Each of these units corresponds to an input or output group, and implements the homonymous scheduling function.

Input to the request function are N arrays, one for each input port. The array elements store the destination IDs and the memory addresses of packets eligible for service at the corresponding input buffer. They are logically connected in a double-linked list, with the head pointing at the oldest packet.

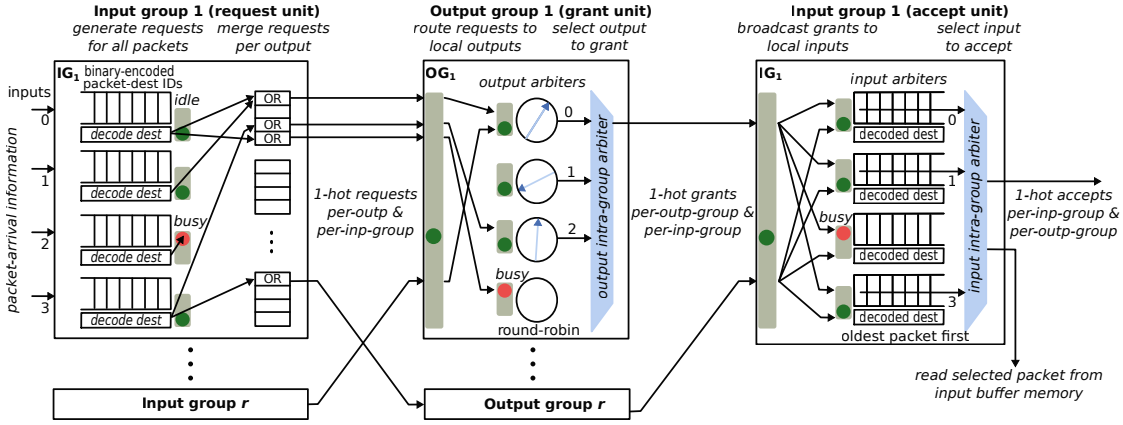


Fig. 2 SCOC scheduling functions for $m = 4$. In SCOC, the output intra-group arbiters implement the oldest-link-first (OLF) policy, and the input intra-group arbiters the random-selection policy. The output arbiters are round-robin (shown are the next-to-server pointers), and the input arbiters, oldest-cell-first (OCF).

A scheduling cycle begins with a request phase. (In parallel, the grant and the accept units are also working, but on requests issued in the preceding and the penultimate clock cycle, respectively.) Every input decodes the destination ID of all packets in its array, and propagates the resulting bits to their corresponding *request merger*. The per-destination request mergers OR the requests coming from the local inputs. Effectively, every input group presents one single-bit request for every destination; in total, there are $r \times N$ request wires in the scheduler.

The requests issued are available to their grant units in the next clock cycle; therein, they are propagated to their corresponding *output arbiter*. Every output arbiter may be requested by up to r input groups; it selects one among them using a next-to-serve, *round-robin (RR)* pointer. However, a grant unit (corresponding to an output group) can issue at most one grant per clock cycle. An *output intra-group arbiter* selects one local (output) grant, and issues it to the corresponding input group. We use an *oldest-link-first (OLF)* policy for this task. OLF is implemented as LRG in [31], choosing the least recently selected link. There are $r \times N$ grant wires in the system.

In the next clock cycle, up to N output grants may arrive at an accept unit (input group). These grants do not specify a particular local input port and are thus broadcast to all. Each *input arbiter* uses the *oldest-cell-first (OCF)* policy to accept one grant. One or more inputs may concurrently accept grants that come from one or more outputs. An *input intra-group arbiter* selects one of among these inputs using the *random* policy [3]. The accept unit then removes the corresponding packet entry from its request array and issues a *packet read command* to its memory buffer. The first word of the matched packet will be transferred after two clock cycles.

Finally, the accept unit asserts an accept signal that terminates to the selected grant unit. As in *i*SLIP, the accept signal moves the next-to-serve pointer of the granting output arbiter to one position following the accepting input group. In SCOC, the accept also updates the state of the intra-group arbiter. The grant unit remembers which local output issued the grant in the preceding clock cycle; hence, there is no need for accept signals to specify an output port—they only have to specify an input-output group pair. In this way, we need $r \times r$ accept wires, instead of $r \times N$.

Weightage: Each output arbiter receives one "merged" request from every input group. In this way, input groups with many active ports are treated as a single port—see microbenchmark 5 in Sec. 3. To resolve this, we introduce a *weightage* method, which adds $\log_2(m)$ control wires per input-output group pair. In clock cycle cc, the input groups use these wires to communicate the number of local inputs (minus one if greater than zero) requesting output cc mod m. Each output holds per-input-group weightage registers, which are updated every m clock cycles. After issuing a grant to a *new* input group, the output arbiter latches the corresponding weightage value to a *repeat* variable. While greater than zero, the 'repeat' variable is decremented by one with every grant issued by the output to the *same* input group. If the 'repeat' variable is zero when the output receives an accept from input group i, then we set the next-to-server RR pointer as $((i + 1) \bmod r)$; if the 'repeat' variable is greater than zero, we move the next-to-serve pointer to the granted input group. We decrement the 'repeat' variable on the grant phase rather than wait for an accept: If an input group does not accept, then it is likely that it also has traffic for other destinations; hence, we legitimately decrement its "quota". Also, if one or more of the "first" grants are accepted by input group i, then we set the next-to-serve pointer as $(i + 1) \bmod r$ once the 'repeat' variable becomes zero, without waiting for a last accept.

C. Asynchronous operation

For asynchronous operation, the arbiter of a port or internal link must be quiescent in clock cycles when its resource is busy. For instance, a busy input should not issue requests if a packet generated by a new scheduling cycle will clash with its current transfer or with one that has been scheduled but not yet commenced. We delegate this job to toggle switches (depicted as "traffic lights" in Fig. 2), which filter out scheduling messages during such busy periods.

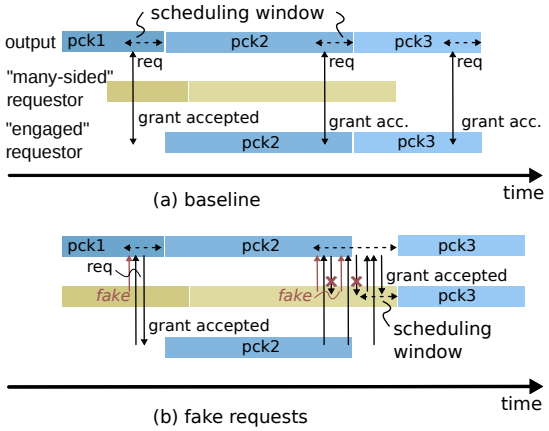


Fig. 3 In (a), the "many-sided" requestor is busy and thus cannot issue requests while the output schedules; effectively, the "engaged" requestor gets all grants. In (b), by using *fake requests*, requestors can override their busy status. The figure assumes that the first grant goes to the "engaged" requestor. However, the second, third and fourth grants are on spot. Although these grants are rejected by the "many-sided" requestor, the output policy is to issue them until one is accepted.

An input port becomes busy when it is matched in the accept phase, an output port when it issues a grant (selected by the intra-group arbiter) in the grant phase. Note that this grant may not be accepted in the next clock cycle, but it would be unsafe to have two outstanding grants. If a first grant is not accepted, an output can issue a second one after two clock cycles. The matched ports may safely revert to "idle" state four cycles (= scheduling + memory read delay) before the last word of the transferred packet enters the interconnect.

Fake requests: The asynchronous scheduling may compromise the fairness of SCOC. In particular, a "many-sided" requestor, be it an input port or an input group that hosts multiple connections, has a disadvantage against requestors with only one connection: When the output port arbitrates, the "many-sided" requestor may be busy, hence it may be bypassed by the next-to-serve output pointer. We overcome this implication by allowing busy units to issue *fake requests*. An example is shown in Fig. 3. Although they may be granted, fake requests cannot be accepted. Also, the output ports cannot distinguish fake from non-fake requests. As with all grants, the grants to busy requestors persist until one is accepted, effectively until the requestor is idle. Note that fake requests may result in wasted output grants. To reduce their number, we allow up to one fake request per input.

D. Timesliced route allocation

Our scheduling obviates the need of separate, per-route, request/grant/accept wires and of per-route arbiters. Instead, it uses a single set of point-to-point wires and arbiters; these are allocated to the per-route operations in a time-division multiplexing (TDM) manner[1].

[1]This route allocation is used for any implementation of the middle-stage switches, be it m separate crossbars or a single m-way TDM one.

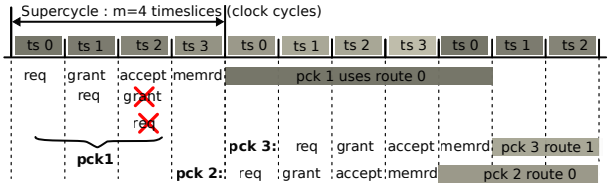


Fig. 4 Timesliced route allocation for $m = 4$ and four (4) independent middle-stage crossbars. Regarding packet 1, the output does not issue a grant in $ts = 2$ because it issued one in the previous cycle—it could have issued one in the next clock cycle, but there was no request to grant; additionally, packet 1 does not issue a new request in $ts = 2$, because it now accepts a grant.

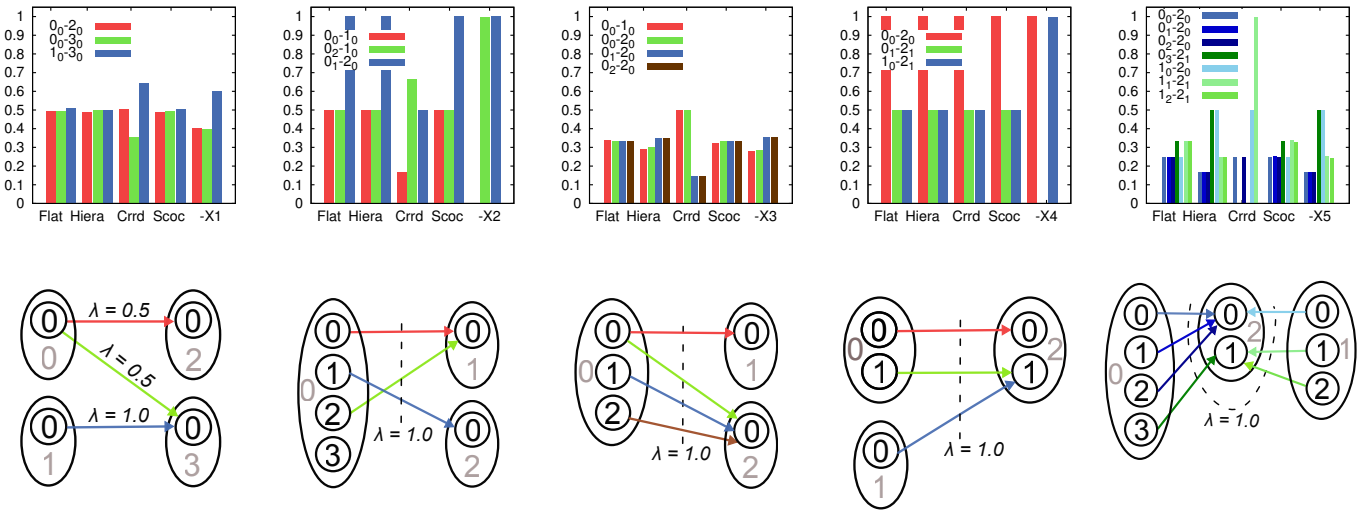
Time is conceptually organized in *supercycles*, i.e., groups of m consecutive clock cycles. Every supercycle consists of m *timeslices*, equal to the number of middle-stage switches. If we enumerate clock cycles with variable cc, then cc mod m yields the index (or *color*) of the corresponding timeslice. As shown in Fig. 4, a supercycle begins at every clock cycle such that $(cc \bmod m) = 0$, and terminates together with clock cycle $(cc + m - 1)$. In the following, the current timeslice is denoted by ts. Also, we may add an integer $n \in [0, m-1]$ to ts, yielding timeslice $(ts + n) \bmod m$.

In SCOC, the route of a packet matches the "color" of the request clock cycle in the corresponding scheduling cycle. Effectively, a grant issued by output group O in timeslice ts attempts a transfer through the middle-stage switch (route) with index $ts + m - 1$.

The input and output groups maintain an m-bit vector, AR, to keep note of their available routes. Output group O issues no grant when $AR_O[ts + m - 1] =$ "busy". Similarly, with *selective requests*, input group I does not issue requests when $AR_I[ts] =$ "busy". This routing mode can yield grossly unfair schedules. To remedy this, we allow input groups to issue "fake" requests on busy timeslices. To reduce the number of wasted grants, i.e., grants issued to busy input groups, inputs may issue fake requests only for their oldest packets, i.e., up to one request each.

Note that *routes are allocated on a per-packet basis*. The route of every packet depends on the stochastic distribution of transfer finish times and on the contention at ports—a packet that is not scheduled in one cycle may end up on a different route. Although the packet route selection is not strictly random, it is approximated as such by algorithm F in Sec. 4.C. When speedup is employed, the transfer finish times are wildly scattered in time. Effectively, combined with a small speedup, our routing strategy behaves close to *random per-packet load-balancing*, overcoming the limitations of static routing schemes [6]. We were not able to find any "bad" traffic permutation for SCOC, and we tested many hundreds, including some well-known ones, like *bit-reverse*, *bit-complement*, *shuffle* and *transpose*.

An example is shown in Fig. 4. Packets 1 and 3 belong to the same connection whereas packet 2 to a different one. As can be seen, the input issues two requests for packet 1, but the output only one grant. The input accepts the grant for packet 1



(a) scenario 1 **(b) scenario 2** **(c) scenario 3** **(d) scenario 4** **(e) scenario 5**

Fig. 5 Fairness microbenchmarks (I). (Top) Per-connection normalized throughputs. (Bottom) Traffic scenarios; in all but scenario 1, connections arrive at full line rate. In these experiments, Flat uses the same speedup as SCOC, i.e., 1.45×.

in ts 2. Packet 1 is transferred through the middle-stage switch 0, which matches the timeslice of the corresponding request. Packet 3 is transferred immediately after packet 1; however it uses a different route—1 vs. 0.

3. SCOC IS AS FAIR AS A FLAT CROSSBAR IS

SCOC is designed to replace flat crossbars in high-performance switch chips. To qualify for the task, SCOC has to maintain the same fairness levels as flat crossbars. We used a series of microbenchmarks to test and improve its fairness. Below, we use them as guides to expose the potential fairness issues and to present how we resolved the latter in our architecture. These microbenchmarks have been reproduced in hardware, validating the fairness properties of our implementation. The results presented below are generated by a C++ model that accurately simulates the implementation of SCOC.

A. Microbenchmarks

In this section, besides the performance of SCOC, we also evaluate *(i)* Flat (synchronous) crossbars with a speedup of 1.45×, running one iteration of *i*SLIP [18], *(ii)* hierarchical crossbars (Hiera) [15], [30] and *(iii)* CRRD [23], a scheduled Clos network, also discussed in Sec.6. In all systems, the buffers per input are shared among the local VOQs [32] and can store up to 16 packets. For CRRD, because the inputs in the same group share their VOQs, we assume that they also share their buffers.

Microbenchmark 1: In our first scenario, shown in Fig. 5(a), two connections, originating from input 0_0 (i.e., port 0 in group 0), target two outputs, 2_0 and 3_0. Output 3_0 is also requested by input 1_0. The throughputs are shown at the top of the chart. Both Flat and Hiera yielded the expected (fair) rates, i.e., approximately 0.5 to each connection. CRRD was unfair in this scenario, assigning a higher rate to input 1_0. SCOC performs identical to Flat. However, to achieve this, we first had to tweak the scheduler. Because scheduling in SCOC is asynchronous, the input (0_0) of two connections may often be busy while one of its output targets (3_0) is scheduling. Effectively, input 1_0 is likely to have an advantage—label $X1$ in Fig. 5(a).

We came up with two ways to overcome this limitation, both of which yield the expected result: *(i)* issuing fake requests from busy inputs; and *(ii)* allow *multiple concurrent transfers per input*. In our implementation, each input buffer can afford reading up to four packets in parallel, in a TDM fashion, i.e., alternating between (words from different) packets in contiguous timeslices. We preferred the second alternative, because, as described in Sec. 4.E, it also improves the small-packet throughput (a similar observation is also made in [21]).

Microbenchmark 2: In our second scenario, Fig. 5(b), we set up three connections coming from different ports in the same input group. One of them targets output 2_0 and the other two target output 0_1. Again, Flat and Hiera yield fair rates. In contrast, CRRD severely penalizes the connection from input 0. SCOC performs as well as Flat does. Label "X2" shows the performance of SCOC using RR instead of random selection at input intra-group arbiters: Because input 1 is served at full rate, when a grant arrives from output 1_0, the next-to-serve RR pointer usually points to input 2; effectively, in our simulations, input 2 gets nearly all grants from output 1_0, leaving input 0 to starve. SCOC resolves this problem using a random arbiter for input port selection. OLF also works fine in this scenario, but it fails in the next one.

Microbenchmark 3: Consider the scenario in Fig. 5(c). Three inputs in the same group target output 2_0. One of them also targets output 1_0. As can be seen, SCOC, using random input

port selection, yields the fair throughputs with an accuracy better than 3%. In contrast, OLF ("X3") treats the two connections that come from input 0 badly. This penalty is not that heavy (18% accuracy), but when we simulated larger input buffers (1K packets vs. 16), OLF nearly starved connection $0_0 \rightarrow 2_0$: Because input 0 also accepts grants from output 1_0, it has the lowest (OLF) priority when a grant arrives from 2_0. With 16-packet buffers, the input immediately drains off buffers once $0_1 \rightarrow 2_0$ starves; effectively, connection $0_1 \rightarrow 1_0$ is also not served, which breaks this destructive loop. In our simulations, with "X3", the two connections from input 0 stabilized at equal, but still suboptimal rates. Hiera and CRRD also failed in this scenario.

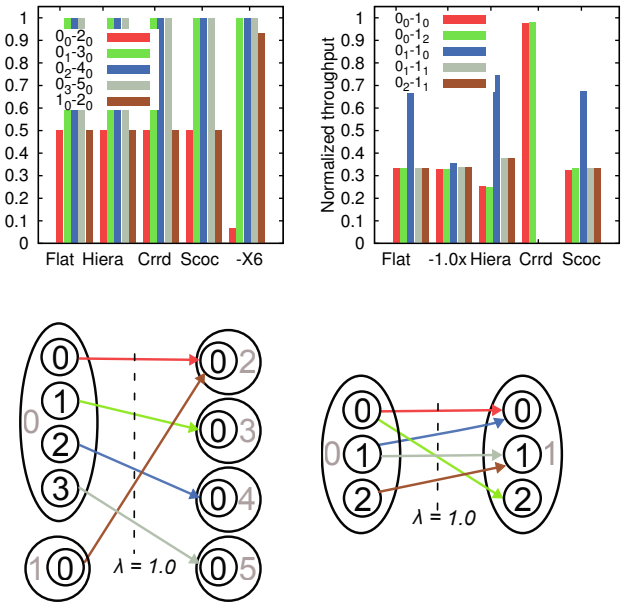
Microbenchmark 4: In our next scenario, Fig. 5(d), three connections from two different input groups target two outputs in group 2. Label "X4" assumes an alternative grant policy for SCOC: First, an output group uses a RR arbiter to select which input group should receive the next grant; second, it picks a local output that can grant the selected input group. At first sight, this policy seems legitimate. However, it can be grossly unfair in the current scenario. Similarly to scenario 2, the problem is that the next-to-serve pointer of the RR arbiter may "bend towards" a particular input group. Here, it usually points to input group 1, because input group 0 is served at full line rate by output 2_0.

SCOC uses a reverse order to allocate grants: First, the RR arbiter at each output port selects one of the requesting input groups; second, the output inter-group arbiter selects one of the outputs using OLF. We also considered RR instead of OLF, but for the conveniently small numbers of ports per group in SCOC ($m \leq 16$), OLF can be implemented at no significant cost and is more precise than RR. SCOC also performs well with random output port selection—see mode 2 in Sec. 4.D. Nevertheless, we prefer OLF because it is more deterministic.

In microbenchmark 3, we saw that OLF at input groups may starve connections when it selects which input port will get the next grant. At input groups, the OLF intra-group arbiter effectively distributes the output grants to competing inputs. In contrast, there is no such scarce resource at output groups: All output ports can issue grants at full speed.

Microbenchmark 5 touches the "Achilles' heel" of most hierarchical schedulers. As shown in Fig. 5(e), two input groups, 0 and 1, target two outputs of group 2. Group 0 has three inputs targeting output 0 and one targeting output 1; group 1 has one input targeting output 0 and two for output 1. As can be seen, Flat and SCOC distribute the rates properly, according to the output fair shares. However, both Hiera and CRRD fail to do so. The same holds also for "X5", i.e., SCOC before applying the *weightage* method. Hiera and CRRD naively merge the requests coming from flows of the same input group, thus compromising fairness. Actually, with CRRD, three flows are starving in Fig. 5(e). We observed that, using more buffers, CRRD can reach the (suboptimal) performance of Hiera and of "X5".

Microbenchmark 6 demonstrates why we avoid *selective requests*. As shown in Fig. 6(a), two connections target destination 2_0. One comes from an almost empty input group (0), whereas the other has three neighbors, targeting different



(a) scenario 6 (b) scenario 7

Fig. 6 Fairness microbenchmarks (II).

outputs. With selective requests ("X6"), the connection coming from the "busy" group cannot get its fair share at the output: When output 2_0 selects a new packet, input group 0 can be busy transferring, thus missing the chance for a grant. In SCOC, the input group issues "fake" requests when it is busy, thus the output will grant it at its round-robin (RR) turn. This grant will be rejected, but the output will not update its next-to-serve pointer until the busy input group becomes available and accepts. Using this mechanism, SCOC achieves fair rates.

Microbenchmark 7 adapts a traffic scenario from [18] by locating all active inputs (and outputs) in the same group. The scenario, shown in Fig. 6(b), stresses *i*SLIP: Without speedup, Flat allocates a suboptimal rate of around 0.33 to all connections (label "1.0×"). SCOC achieves the same performance as Flat *with* speedup. Both Hiera and CRRD deviate from that, with the latter actually starving three connections.

In this section, we presented a subset of the many fairness microbenchmarks we have considered. With proper policies in place, SCOC succeeded in all of them. In contrast, as we demonstrated, Hiera and CRRD failed in several of them.

4. SCOC IS EFFICIENT

SCOC features a rearrangeably non-blocking topology, which it fully exploits using per-packet multipathing. Hence, it has all that it needs to be efficient. However, many packets may compete for a limited number of internal links. The optimal solution for the corresponding *route-allocation* task involves edge coloring in bipartite multi-graphs [12] [4]. Being distributed to per-port and per-group arbiters, the scheduler of SCOC is non-optimal. In this section, we evaluate how well it performs.

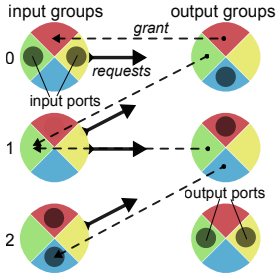


Fig. 7 Example of route assignments for $m = 4$. For the sake of clarity, only three input and output groups are shown (outer circles with colored quadrants). Each quadrant corresponds to a route (middle-stage switch). Filled circles indicate busy ports, sending or receiving a packet in parallel to the ongoing arbitration—their orientation within their group is determined solely by the route they are using.

A. Route allocation: the performance

For a packet to be transferred in SCOC, the corresponding input and output groups must agree on a route available to both. Let $\Re(G)$ denote the set of available routes at input or output group G.

With fake requests, i.e., the default operation in SCOC, input groups do not expose their route availability to output groups. On their part, output groups reserve one available route for every grant they issue, and communicate their selection to the granted input. An input group may receive multiple grants, spanning from 1 to m different routes. It can accept up to m of them, one per route in $\Re(i)$ and per local port, but rejects grants on routes that are busy.

Some examples of route assignments for $m = 4$ are shown in Fig. 7. Input and output groups are depicted with colored circles and routes with quadrants. Filled circles indicate busy ports, occupying their surrounding route (quadrant); their orientation is determined solely by the route they are using.

Input group 0 requests some port(s) in output group 0, which grants one, proposing the "topmost" route (quadrant). This grant is accepted as the topmost route is also available at input group 0. Input group 1 requests output groups 0 and 1, both of which propose the "leftmost" route. The input group can accept only one of these grants. Finally, output group 1 offers the "bottommost" route to input group 2. Because this route is busy for input group 2, this grant is rejected.

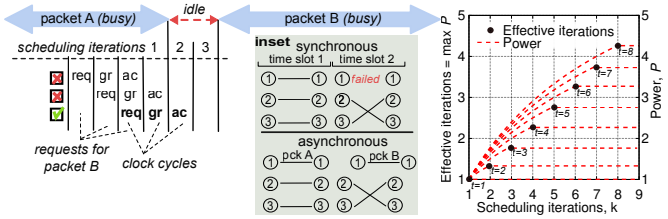
Every non-accepted grant introduces a "bubble" in the scheduling pipeline. SCOC deals with the ensuing throughput penalty by

1) using a small internal speedup, s,
2) iterating the scheduling handshake, and
3) adhering to good schedules.

As we discuss below, mechanisms *(2)* and *(3)* are implemented implicitly by the workings of the scheduler, and incur zero hardware cost.

B. Scheduling iterations

We name *scheduling cycle* the sequence of request, grant, accept and memory read operations that are associated with



(a) (b)

Fig. 8 (a) Packet B could have been scheduled in the first scheduling iteration, but instead it succeeds in the third one, i.e., two clock cycles later. For comparison (as shown by the inset), with synchronous crossbars, an analogously suboptimal schedule costs one or more time-slots (packet times) of idle time. (b) Effective iterations for various packet transfer times, t.

the input and output ports of a particular packet and which are executed in contiguous clock cycles. A successful scheduling cycle sets up the transfer of a packet from its input buffer to the target output. Iterative crossbar schedulers repeat the request-grant-accept handshake to augment the crossbar configuration of a single time-slot with additional port pairs [3] [18]. An analogous effect takes place in SCOC, but instead of packing more operations into a specified time budget (i.e., a time-slot), SCOC spreads them in time.

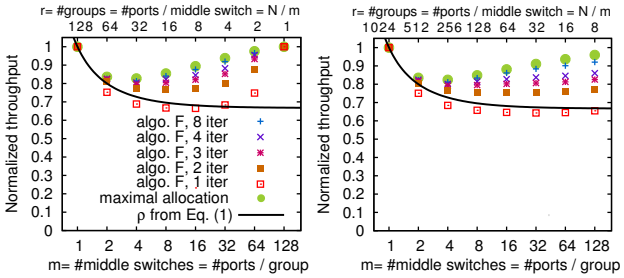
As shown in Fig. 8(a), packets in input buffers may push requests into the scheduling pipeline of SCOC in every clock cycle until they are transferred. Because the independent port-level arbiters may produce suboptimal configurations, a packet B may not be transferred after a first scheduling cycle although the corresponding ports stay idle. In this case, B may be scheduled in one of the next scheduling cycles, which finish after i (≥ 1; 2 in the figure) clock cycles. We can think of these contiguous scheduling cycles as extra *scheduling iterations*. If B_s is the size of packet B and w the width of the datapath, both measured in bytes, there will be i idle clock cycles followed by $t = \lceil \frac{B_s}{w} \rceil$ busy ones. Obviously, the overhead is negligible for sufficiently large packets.

We want to estimate the *effective iterations*, taking into account the "idling time" of iterations relative to the packet transfer time t. The first iteration comes for free, but every subsequent iteration consumes one clock cycle. Assuming that i iterations produce an aggregate positive work (force) i at a pace (velocity) $\frac{t}{t+i-1}$, then the product of the two measures *power* $P_t(i) = i \cdot (\frac{t}{t+i-1})$. The work produced levels off after $i = t$ iterations. We define the effective iterations as $\kappa_t = \max_i P_t(i)$.

Figure 8(b) plots the power $P_t(i)$ and the effective iterations κ_t as a function of t. As can be seen, for $t = 1$, $\kappa_t = 1$. For $t = 2$, κ_t is between $1 - 2$, and between $4 - 5$ for $t = 8$.

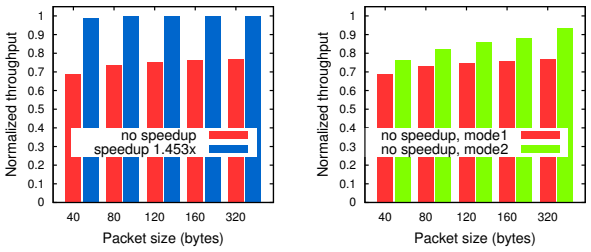
C. Iterating to a maximal route allocation

Next, we quantify how iterations improve the non-optimal route assignments of SCOC. We consider port-level permutation traffic to eliminate the effects of port contention—this does not remove the contention among groups.



(a) $N = 128$ (b) $N = 1024$

Fig. 9 Throughput of routing algorithms F and F' (maximal allocation) for $N = 128$ and 1024 ports, and varying number of middle-stage switches m.



(a) $N = 128$ (mode 1) (b) $N = 256$

Fig. 10 Throughput of SCOC under random-permutation traffic for $m = 4$, and varying packet size.

Let π_o denote the sourcing input of output o, OG_o the output group of output o, and IG_{π_o} the input group of input π_o. Consider the following algorithm, F, which assigns routes to connections $\pi_o \rightarrow o$, $o \in [0, N-1]$.

init. Mark that the set of matched connections $S = \varnothing$. Also, initialize $\Re(IG_g) = \{x \in [0, m-1]\}$, $\Re(OG_g) = \{x \in [0, m-1]\}$ for all groups g, and select a random ordering of outputs ϕ.

1. Iterate through steps 2-4 for k times.
2. Visit all unmatched outputs in the order dictated by ϕ.
3. For each visited output o ($\pi_o = i$), select route m' from $\Re(OG_o)$ uniformly and at random,
4. If $m' \in \Re(IG_i)$, then $S = S \cup (i \rightarrow o)$, $\Re(IG_i) = \Re(IG_i) \setminus m'$, and $\Re(OG_o) = \Re(OG_o) \setminus m'$.

Steps 2-4 emulate the routing assignments in SCOC: Output o knows that the granted packet can reach it from the middle-stage m', but is totally unaware of the situation at the first leg of the route, because input groups may issue requests on busy timeslices. Effectively, input i may reject the output grant. Similarly to SCOC, algorithm F uses iterations to ameliorate the allocation of routes, without modifying already routed connections.

Incapable of backtracking, routing algorithm F converges to a *maximal route allocation*. A route allocation is maximal if, for every pending connection, the sets of available routes at the corresponding input and output groups are disjoint. Routing algorithm F can be adapted to compute a maximal match from the first iteration: Modify step 3 so that every output randomly selects a route m' from $\Re(OG_o) \cap \Re(IG_i)$, immediately continuing with the next output in ϕ if $m' = \varnothing$. We name this routing algorithm, first proposed in [11] [4], F'. In SCOC, this is analogous to the *selective-request* mode, where input groups issue requests only when they are available. This may improve the throughput, but, as seen in Sec. 3, can be unfair.

Figure 9(a,b) compares SCOC route allocation, F, with maximal route allocation (F'). The throughput of these algorithms is computed after they terminate as $|S|/N$. Each reported value is the average throughput of 20,000 randomly selected permutations [16]. We also plot the expected throughput of F with one iteration, ρ, which is estimated by the following formula (for brevity, we do not present its derivation here):

$$\rho = \frac{1}{1 + \frac{m-1}{2 \cdot m}} \quad . \tag{1}$$

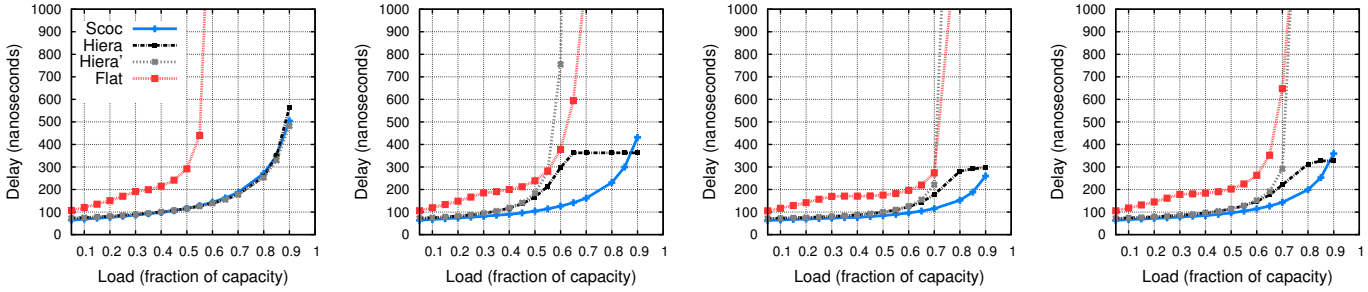
As can be seen, for $m \ll N$, Eq. (1) agrees closely with the results obtained by running algorithm F. For these practical values of m (≤ 16), the throughput is independent of the switch size N. When $m = 4$, it is approximately 0.69 for $k = 1$, and builds up to 0.77 for $k = 2$ and to 0.80 for $k = 3$ iterations. However, there is practically no gain from additional iterations, as we are already pretty close to the maximal performance.

It is interesting to see what happens when $m = N$. In this case, we have N-port input and output groups, and N single-port switches in the middle. As shown in Fig. 9(a), for $m = N = 128$, the throughput is full from the first iteration, because there are no routing conflicts. The same also happens at the other end of the spectrum: When $m = 1$, the topology becomes a flat crossbar (single-port groups). But already as $m = 2$, routing conflicts appear, pulling down the performance of both F and of maximal route allocation.

In Fig. 10(a), we present the throughput of SCOC for $m = 4$, tested here under random permutation traffic using our C++ switch model. Each data point is the average taken from 100 randomly selected permutations. We repeated these experiments for $N = 256$ and 512, and the performance did not change, as also anticipated by the above-mentioned results.

The datapath width is 40B, as in our implementation. Therefore, 40B packets are transmitted in one, 80B in two, and 320B in eight clock cycles ($t = 8$). From Fig. 8(c), the effective iterations in SCOC are ($\kappa_t =$)1, between $1 - 2$, and close to 4, respectively. The corresponding throughputs *without speedup* are 0.687 and 0.73, and 0.77, which match the results of algorithm F for $m = 4$ in Fig. 9. In contrast, when using a marginal speedup of $s = 1.45$, the throughput is close to full for all packet sizes examined.

Although we present here the average performance for many permutations, when using speedup, the throughput was full in *all* permutations we examined. Essentially, SCOC implements a per-packet load-balancing scheme, and its performance does not depend on the orientation of the traffic pattern.



(a) Uniform (b) Partitioned (c) Diagonal (d) Logdiagonal

Fig. 11 Load-delay performance of SCOC, Flat (without speedup) and Hierarchical crossbar for various traffic patterns.

D. Adhering to good matches

Figure 10(b) plots the throughput of SCOC *without speedup* for two different modes of operation. Mode 1 is the default mode that uses the *Oldest Link First (OLF)* policy to select among the output subports that can concurrently issue grants. Mode 2 uses the random policy for the same task. As can be seen, the throughput of mode 2 exceeds that of mode 1 substantially. This is explained below.

With asynchronous scheduling, when a packet transfer finishes, the corresponding input and output ports become available at the same time, while other ports may be busy transferring; hence, the now released ports are likely to match again [25]. In our simulations, we observed that the fewer the idle ports in SCOC, the longer will the existing port pairings persist. Thus, if the scheduler finds a configuration with many busy ports, it is likely that it will hold on to it until the relevant VOQs drain away. In the experiment of Fig. 10(b), mode 2 unlocked this potential more frequently than mode 1 did.

A downside of such asynchronous scheduling is described in [25]. By drifting towards existing port pairings, asynchronous scheduling can induce prolonged starvation under pathological scenarios. The short input buffers of SCOC, its internal speedup, and an aging mechanism that is used as a last resort, eliminate this possibility.

E. Synthetic traffic

Next, we measure SCOC's delay and throughput performance under various synthetic traffic patterns. Our computer simulation model follows SCOC implementation for $m = 4$.

The packet delay is measured from the time that the first bit arrives at the switch input until the time it departs at its output. The network is warmed up before collecting measurements. Each computer simulation is repeated at least five times, for different random seeds. Below we present the average performance value of interest from these five simulations. Unless otherwise noted, we assume a 128-port switch, with 25 Gb/s links. Our baseline packet size is 288B[2].

Figure 11 presents the delay versus load for SCOC, Flat (without speedup) and Hiera. Flat has the same buffers as SCOC (16 packets per input and output port), and Hiera is a 128-port hierarchical crossbar switch implemented using 256 8-port subswitches [15], [30]. If Hiera's subswitches have buffers for b_s packets per input each, then each Hiera input can store a total of $16 \cdot b_s$ packets. Thus, for a fair comparison, (i) we set $b_s = 1$, in which case Hiera has the same input buffers as the other systems, and (ii) we also consider *Hiera'*, which has ($b_s =$)$16\times$ larger buffers.

SCOC implements cut-through, with a fall-through packet latency of 61 ns: Approx. 15 ns per direction for SERDES and clock domain crossing, and 15 cycles at 454 MHz for crossbar and link receive/send logic. The packet delay *inside* the Hiera chip is approximately 25 cycles at 800 MHz [15], i.e., 32 ns, but this does not include the chip interfacing delays. We assume that the SERDES and packet processing delays are small, yielding the same fall-through latency as SCOC (61 ns). Flat uses store-and-forward; thus, the delay of 288B packets at 25 Gb/s links cannot get below 92 ns.

Figure 11(a), for uniform traffic, shows that Hiera and SCOC perform identically, both tolerating a load up to 0.9—although they can both go higher, we decided to not present data for very high loads[3]. Flat performs considerably worse, (i) because it does not have speedup, and (ii) because the *i*SLIP algorithm fails to desynchronize when the input packet buffers can hold only 16 packets each: For successful desynchronization, inputs must hold packets for all destinations. Effectively, the throughput of Flat is limited at around 0.61, and, as can be seen in the figure, its delays are much higher at even lower loads.

Having examined all-to-all traffic, we now consider a *partitioned workload*, wherein end nodes, e.g., servers or cores, work in groups. In Fig 11(b), we partition the 128 switch ports into 16 groups of eight, and apply uniform, all-to-all traffic within every group. As can be seen, both Hiera and Hiera' saturate close to 0.65 load. This happens because we purposely aligned the boundaries of the groups with those of the subswitches [15]. Because it has larger buffers, the delay of Hiera' caps that of Hiera close to and after the saturation point. In contrast, the performance of SCOC is not affected by the

[2]Chosen as an adequately large value that is also not necessarily a multiple of the datapath width: In SCOC, such large packets performed identically, despite their exact alignment. In our server-rack fabric, this is also a very frequent size of the segments (internal packets) that carry the payload of 1500B Ethernet frames.

[3]The simulations sometimes take too long to converge at high loads; moreover, long-lasting overloads are not tolerated in operational units.

partitioning. A (folded-)Clos network, with arbitrary routing, may perform badly depending on the orientation (principal permutation) of the traffic pattern [6]. However, this is not the case for SCOC, thanks to its per-packet load balancing.

Figures 11(c,d) examine non-uniform traffic patterns. With *diagonal traffic*, every input i steers two thirds of its load to output $(i+1)$ mod N and the remaining one third to output i mod N. With *logdiagonal (or power2) traffic*, every port i sends to all other ports, but sends twice as much traffic to port i as to $(i+1)$ mod N, and so on. As can be seen, SCOC still outperforms Flat and delivers 15-20% higher throughput than Hiera.

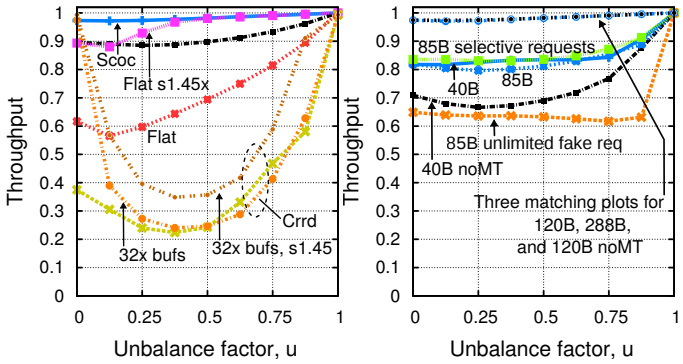
Next, we use an unbalanced traffic model from [29]: Every end-node i is associated with a second end-node, i.e., its primary sink, and with a third end-node, i.e., its primary source. Let *sink(·)* be the permutation that maps every end-node to its primary sink. The results are present in Fig. 12. Along the *x*-axis, the traffic pattern changes gradually: For point $u = 0$, it is uniformly destined, for $u = 1$, it is directed (permutation), and for the values in between, there is a variable mix of directed and uniform traffic. In this test, *sink* is the identity permutation.

In Fig.12(a), we compare SCOC with Flat and CRRD. As can be seen, SCOC outperforms Flat even when the latter uses a speedup $s = 1.45\times$. This happens because, as discussed in Sec. 4.B, the scheduling pipeline of SCOC implicitly implements multiple iterations for large packets, whereas that of Flat does not. As mentioned above, the throughput of Flat with no speedup is limited under uniform traffic ($u = 0$) because it lacks adequate buffers. The same happens to CRRD, but here the throughput is even lower. As can be seen, when we increased the buffers in CRRD by a factor of $32\times$, its throughput under uniform traffic got almost full. However, because of the sharing of the input buffers, its throughput is really low for intermediate values of u. We also tested the throughput of CRRD with large buffers and a speedup equal to that of SCOC ($1.45\times$). As can be seen, its throughput can still get more than $2\times$ worse than that of SCOC.

Figure 12(b) presents the throughput of SCOC as a function of the packet size. We examine 40B, 85B, 120B and 288B packets, which are frequent segment sizes in our server-rack fabric. With the exception of 85B packets, the performance of SCOC generally increases with the packet size. 85B packets are padded in 120B ones and transmitted in three clock cycles ($t = 3$ words), hence introducing a slowdown of $85/120$; however, because they are larger, they achieve almost the same performance as 40B, despite the slowdown.

For 85B packets, we test the *selective-request* mode ("X6"), which offers a marginal throughput benefit over SCOC, because it does not issue fake requests. We also present the performance of SCOC if we allow an *unlimited number of fake requests*—SCOC allows only one per input. As can be seen, the performance is affected by these many fake requests, because many more grants are rejected.

The capability of *multiple transfers per input* is the main performance benefit gained by having a single 4-way TDM crossbar in place of four independent crossbars. In Fig. 12(b), we examine designs that cannot afford multiple transfers



(a) Various systems: 288B. (b) SCOC: varying packet size.

Fig. 12 Throughput under uniform ($u = 0$), unbalanced ($0 < u < 1$), and directed ($u = 1$) traffic.

per input (label noMT). For 40B packets (i.e., single word-transfers), the throughput loss is significant, going from 0.8 to 0.65; however, for large packets, e.g., 120B, there is virtually no performance penalty.

Not presented so far are results for mixed-size packets. We have examined a bimodal packet-size distribution, consisting of 90% 40B and 10% 288B packets. The throughput of SCOC exceeded 0.92 for any unbalance factor $u < 1$, and was full for $u = 1$. We have also tested SCOC using flow (message) size distributions from operational datacenters [2]. In these tests, SCOC outperformed hierarchical and flat crossbars with equal speedup.

Scalability: We have repeated many the above-mentioned experiments for 256- and 512-port switches using $m = 4$. The performance of Flat and of CDDR depends on the radix of the switch, worsening with more ports. In contrast, the performance of SCOC is nearly independent of the switch size.

Many non-TDM middle-stage switches ($m > 4$): The results presented so far assumed a single 4-way TDM middle-stage crossbar. The performance of SCOC does not change considerably if we use four independent middle-stage crossbars. The only performance benefit of the TDM crossbar is the increased memory read throughput (multiple transfers per input) that comes with it. As shown in Fig. 12(b), lacking this extra "input speedup" affects the throughput by as much as 15% under small-packet workloads. However, this penalty can be mitigated by slightly increasing the frequency, f, and hence the speedup of the switch.

Furthermore, the timesliced route allocation of SCOC introduces some non-negligible bandwidth overhead, which increases with m: The scheduler can allocate a link (route) l on a packet transfer at most once every m cycles; thus, link l will idle for packet transfers that terminate before completing an integer number of supercycles. Additional analysis and computer simulations—not presented here for brevity—suggest that the overhead can be managed by increasing the frequency of the chip, f, and decreasing the datapath width of the individual crossbars, w: A 256-port SCOC switch, for $m = 16$, $w = 4$B and $f = 1.56$ GHz, offering a speedup of 2 for 25 Gb/s links, achieves close to full throughput under both uniform and unbalanced traffic for packet sizes as small as 64B. This

result is *without* multiple transfers per input—instead, busy input ports issue fake requests for their head-of-line (HOL) packet.

F. Experiments in fat-tree networks

A separate C++ model of our SCOC chip was developed and integrated into our performance model of a server-rack fabric [7]. This is a buffered fat-tree fabric that segments network packets into variable-size units, and forwards the latter using segment-level multipathing. Still in the design phase, we extensively tested the server-rack fabric under many workloads, e.g., micro-bursts, all-to-all and many-to-one (fan-in) communication, standard switch benchmarks, etc, for various packet (and thus segment) sizes. Whenever we ran into an unexpected performance issue, we tested the spines by replacing the SCOC module with that of a flat VOQ crossbar, *using a speedup of two*, to examine the impact. In all cases, the problem was located somewhere else in the design. SCOC was never found to undermine the performance of the system. These tests further validate that SCOC performs virtually identical to efficient flat crossbars when employed in large networks.

5. The SCOC switch chip

We have implemented the SCOC architecture in a 32 nm ASIC. The SCOC chip is used in a spine-leaf server-rack fabric, providing dense connectivity among leaf nodes. Following the current trends in datacenter networks, SCOC has small input (16-packet) and output (12-packet) buffers; packets can be of any size up to 320B. Additionally, SCOC implements a preferable VOQ organization—not presented here for brevity—, with a cost that grows with the number of packets per input instead of the much larger number of switch ports.

The SCOC chip supports 136 bidirectional ports running at 25 (28 raw) Gb/s links. Parameter $m = 4$, therefore there are $r = 34$ input and output groups (or *quads*). Instead of four 34-port middle-stage crossbars, we have implemented a single but four times wider 34-port crossbar. This is a 40-byte crossbar that implements four (4) slower crossbars in a TDM fashion: In clock cycle cc, it carries payload words from packets that would be routed through the middle-stage switch cc mod 4. Effectively, a packet of t words is transmitted in $4 \cdot t$ cycles.

The chip actually contains *two* 136*-port SCOC switches*, one for data (payload) traffic and one for *end-to-end (ETE)* control traffic, e.g., requests, grant-credits and acknowledgments. These ETE messages have a fixed size of 10B. Similarly to the data crossbar, the ETE crossbar groups the ports in quads, but has a single, 10-byte-wide, middle-stage switch. It also has a slightly simpler hierarchical scheduler, allowing arbitration only to the two oldest packets per input. The ETE input buffers can hold up to 256 messages.

The floorplan of the chip is depicted in Fig. 13. There are 136 high-speed serial (HSS) I/O cores in the periphery of chip. The data crossbar and the ETE crossbar, as well as their schedulers, are mainly located in the inner area. The data (ETE) crossbar has a size of 2.5×2.5 (1.2×1.2) mm^2, and its

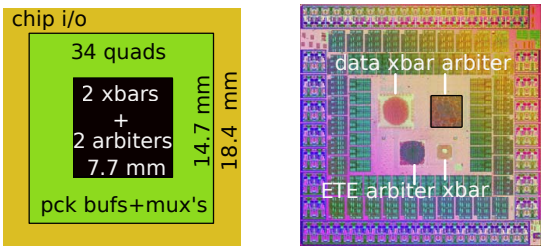


Fig. 13 Floorplan (left) and picture (right) of the 136×136×25 Gb/s SCOC switch chip.

scheduler of 2.6×2.6 (2×2) mm^2; 34 (1.6×1 mm^2) *quads* are positioned around them, each responsible for the queuing and the switching functions within an input and output group.

The chip has a die of 18.4x18.4 mm^2, uses 45 million gates (20% are for test and clock circuity), 11 metal layers, runs at 454 MHz, and consumes around 150 Watt. Of these 150 W, 130 W is consumed by the HSS cores, or about 1 W per link (approximately 1 W / 28 Gb/s = 36 pJ/bit). The entire chip flow, including synthesis, test logic insertion, placement, clock insertion, PD re-synthesis, routing, and timing closure, took six weeks. In contrast, synthesizing the data crossbar arbiter on an 8-core (190 GB) workstation was possible in only 36 h.

The SCOC chip provides a speedup of 1.45×. This speedup was selected early in the design phase after evaluating the expected efficiency of our routing: As described in Sec. 4.C, for $m = 4$, the throughput of SCOC without speedup can be as low as 0.70; therefore, a speedup of 1.45× remedies the routing inefficiencies. We did not push our EDA tools to achieve a higher frequency at that time; however, timing was always a concern as we added new features later.

6. Discussions & related work

A. Design trade-offs

In this section, we discuss some of the trade-offs that emerged while designing SCOC.

Utility vs. guaranteed fairness: To the best of our knowledge, the present paper makes the first systematic attempt to build Clos networks having the fairness properties of flat crossbars. By resolving all fairness issues that we discovered in our architecture and showing that state-of-the-art designs and chip implementations are exposed to many of them, we achieved an important milestone. What is missing from our study is *deterministic* guarantees that we have covered all possible fairness issues. Such analytical attempts typically involve impractically expensive designs [8]. In this paper, we aimed at a feasible, cost-effective implementation and covered the most central fairness issues. In Sec. 3, we presented a selected subset of the fairness tests we examined. We have performed many more tests, all of which validated our architecture.

Adversarial traffic: The following adversarial traffic pattern can compromise the fairness of SCOC. If an input queue fills up with packets heading to a congested switch output, it will exert flow control that can impede the flow of unrelated, non-congested packets coming from the same input. However, this unpleasant behavior is equally valid for flat crossbars. In the absence of end-to-end flow control, congested flows can induce

saturation trees, with unwanted consequences. Our server-rack interconnect uses end-to-end credits, Ethernet QCN at ingress leaf nodes, and segment-level spraying across leaf-spine links [7] to prevent (or limit the duration of) congestion episodes inside the SCOC switches (spine nodes).

How relevant is fair service to real-word workloads? Long-lived persistent connections, such as those examined in Sec. 3, are typical in datacenters [2] and in HPC interconnects. Fair flow service can reduce the tail of flow completion times and improve the performance of latency-sensitive scale-out applications.

Small vs. large buffers: Following a common practice in HPC interconnects [30], high-performance datacenter networks use small in-fabric buffers to reduce the packet delays [33], [2]. In our server-rack interconnect, we use small buffers in spines switches (SCOC) to prevent large in-fabric backlogs, and larger buffers in leaf nodes. Thanks to credit-based flow control between spine and leaf nodes, the buffers of SCOC are backed up by the larger (leaf) buffers, which can segregate flows into network-level VOQs. HPC routers also typically have small on-chip buffers [14].

B. Comparison with state of the art

High-radix switches have attracted renewed attention since Kim et al. quantified the benefits of many "slim" channels per chip over few "fat" ones [15]. As an answer to the increasing complexity and to the poor performance of high-radix input-queued switches, Kim et al. proposed a hierarchical crossbar switch (Hiera), which was later embodied in *YARC* [30]. YARC, implemented by Cray, has the same number of crosspoints but provides higher performance than flat crossbars. Also, it is inherently amenable to a hierarchical EDA flow. Later, the OBIG project by SUN exploited a similar hierarchical design for ultra-high-radix switches enabled by proximity communication [10]. These designs require many small memory units and are inefficient under the partitioned workloads that are common in cloud computing environments and in CMPs. Furthermore, Hiera, despite its higher speedup (8-16$\times$ vs. 1.45), performs worse than SCOC under unbalanced traffic: Because of extensive buffer partitioning, the buffers in Hiera can be underutilized in some traffic patterns. Hiera also failed in some of our unfairness microbenchmarks.

The cost advantages of Clos over hierarchical and flat networks are quantified in [1], where Ahn et al. propose a *buffered* folded-Clos network for high-radix switches. Buffered networks require more (small) buffers than SCOC, and are susceptible to saturation trees. In addition, unless they perform packet-level multipathing (correcting also packet reordering), they are vulnerable under arbitrary permutation patterns. For instance, in [1], the throughput of the proposed network is half of the maximum possible under some permutation patterns. In contrast, using packet-level load balancing, the performance of SCOC does not depend on the spatial orientation of the traffic pattern.

The results of SCOC suggest that, despite being less effective for off-chip networks [24], [5], *a bufferless on-chip fabric is more attractive than a buffered one.* By concentrating the memories at network ports, a bufferless Clos enables denser designs and packet-level multipathing without out-of-order packet delivery. Although Clos networks are generally considered hard to schedule [4], we demonstrated an efficient and scalable scheduler that fixes all fairness issues involved. Together with a small on-chip speedup, SCOC meets the performance levels anticipated by high-end switching nodes.

Bufferless on-chip networks are also building up momentum in NOCs for CMPs [22], [19]. Their benefits include reduced power consumption and silicon area as well as congestion avoidance. However, lacking a global scheduler and contention-resolution buffers inside the network, they are left with deflection routing or packet dropping, which perform sub-optimally from a bandwidth point of view. Coupled with the inherent inefficiencies of the mesh and torus networks for which they are proposed, these alternatives cannot provide the performance levels required by high-radix switches.

Oki et al. use per-packet load-balancing in CRRD, i.e., a scheduling scheme for off-chip, memory-space-memory Clos switches [23]. CRRD depends heavily on pointer desynchronization, the performance of which drops dramatically as we increase the number of ports or when traffic becomes unbalanced [5]. As we have further shown in this paper, CRRD has many unresolved fairness problems, and its performance drops as we decrease the buffers per input.

The Swizzle-Switch [31], by Sewell et al., integrates the arbitration inside a datapath with O(N^2) crosspoints. Because of its gradual contention resolution, the Swizzle-Switch has largely unknown fairness properties. Passas et al. implemented an 128-port flat crossbar for 10G links with a speedup of two, showing reduced area numbers [26]. However, their solution is highly customized, reducing the flexibility of the design and increasing its turn-around time. In contrast, SCOC is amenable to fully automated design flow, while being inherently more scalable. Finally, another recent switch chip from IBM, the PERCS Hub-Chip [28], has almost the same internal bandwidth as SCOC. The PERCS Hub-Chip is implemented in 45 nm, occupies 582 mm^2, and uses 13 metal layers. Being a processor hub-chip, it supports links with varying speeds: 31 Local (L) links, 16 Remote (D) Links, 4 POWER7 QCM Connect links, and 3 PCIe links.

7. Conclusions & future work

We presented SCOC, a scalable bufferless Clos architecture that can readily replace the flat crossbars in high-radix switch nodes and on-chip networks. Although buffered fabrics are preferable off-chip, our design demonstrates that the situation is reversed inside the chip: Bufferless Clos networks enable packet-level multipathing, thus avoiding the throughput limitations of static routing. This, coupled with cheap on-chip speedup, allows SCOC to outperform more expensive designs, such as hierarchical and flat crossbars.

Designing SCOC, we made a significant effort to characterize and improve its fairness levels, something that is missing from previous studies. Despite its hierarchical contention resolution, SCOC appears to have the same fairness properties as flat crossbars, while being scalable to an order of magnitude larger port counts.

Whereas it is hard to build high-radix crossbars, the switch chips of today are mostly I/O limited. Emerging optical link technologies promise to deliver many thousands of high-speed links per chip [20]. This much expected technological breakthrough will accelerate exascale-class systems, which seek to sustain the I/O byte-per-flop ratio of today while increasing the capacity of compute nodes by multiple orders of magnitude [13]. Once optics make it to chip boundaries, crossbar chips will become core limited again. The design of SCOC suggests that it may be possible to distribute the switching logic (crossbar tiles) over multiple 3D-stacked chip dies, while maintaining the performance levels of single-stage, single-stack crossbars. SCOC also presents a competent alternative to (low-performance) mesh networks and to (hard-to-scale) crossbars for on-chip networks used in chip multiprocessors.

So far, the design of SCOC has been validated by a fully-operational chip that implements two SCOC switches that share 136 ports at 25 Gb/s. The entire chip die, including the I/O circuity, occupies just 18.4×18.4 mm^2.

8. Acknowledgments

The authors would like to thank the reviewers of HPCA 2015 as well as Charlotte Bolliger for their comments.

References

[1] J. H. Ahn, S. Choo, and J. Kim, "Network within a Network Approach to Create a Scalable High-Radix Router Microarchitecture," in *Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2012, pp. 1–12.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz *et al.*, "DCTCP: Efficient Packet Transport for the Commoditized Data Center," in *Proc. ACM SIGCOMM*, New Delhi, India, Aug. 2010.

[3] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High-speed Switch Scheduling for Local-Area Networks," *ACM Transactions on Computer Systems*, vol. 11, no. 4, pp. 319–352, 1993.

[4] H. J. Chao, Z. Jing, and S. Y. Liew, "Matching Algorithms For Three-Stage Bufferless Clos Network Switches," *IEEE Communications Magazine*, vol. 41, pp. 46–54, Oct. 2003.

[5] N. Chrysos and M. Katevenis, "Scheduling in Non-Blocking, Buffered, Three-Stage Switching Fabrics," in *Proc. IEEE INFOCOM*, Barcelona, Spain, Apr. 2006.

[6] N. Chrysos, F. Neeser, M. Gusat, C. Minkenberg, W. Denzel, and C. Basso, "All Routes to Efficient Datacenter Fabrics," in *Proc. INA-OCMC*, Berlin, Germany, Jan. 2014.

[7] N. Chrysos, N. Neeser, R. Clauberg, D. Crisan, K. Valk, C. Basso, C. Minkenberg, and M. Gusat, "Unbiased QCN for Scalable Server-Fabrics," *IBM Research Report, RZ3880*, October 2014. [Online]. Available: http://domino.research.ibm.com/library/ cyberdig.nsf/papers/EF857B498A21B3EF85257D6A002E2235

[8] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, "Matching Output Queueing with a Combined Input/Output-Queued Switch," *IEEE Journal Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, Jun. 1999.

[9] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[10] H. Eberle, A. Chow, B. Coates, J. Cunningham, R. Drost, J. Ebergen, S. Fairbanks, J. Gainsley, N. Gura *et al.*, "Multi-Terabit Switch Fabrics Enabled by Proximity Communication," in *IEEE Hot-Chips*, 2007.

[11] K. Y. Eng, M. J. Karol, and Y.-S. Yeh, "A Growable Packet (ATM) Switch Architecture: Design Principles and Application," *IEEE Transactions on Communications*, vol. 40, no. 2, pp. 423–430, 1992.

[12] H. N. Gabow and O. Kariv, "Algorithms for Edge Coloring Bipartite Graphs and Multigraphs," *SIAM Journal on Computing*, vol. 11, no. 1, pp. 117–129, 1982.

[13] M. Haney, R. Nair, and T. Gu, "Chip-scale Integrated Optical Interconnects: A Key Enabler for Future High-Performance Computing," in *SPIE OPTO*. International Society for Optics and Photonics, 2012.

[14] J. Kim, W. J. Dally, J. Dally, and D. Abts, "Adaptive Routing in High-Radix Clos Networks," in *Proc. ACM/IEEE SC*, 2006.

[15] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, "Microarchitecture of a High-Radix Router," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 420–431.

[16] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[17] J. Koehl, D. E. Lackey, and G. Doerre, "IBM's 50 Million Gate ASICs," in *Proc. Asia and South Pacific Design Automation Conference*. ACM, 2003, pp. 628–634.

[18] N. McKeown, "The iSLIP Scheduling Algorithm for Input-Queued Switches," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, 1999.

[19] G. Michelogiannakis, D. Sanchez, W. J. Dally, and C. Kozyrakis, "Evaluating Bufferless Flow Control for On-Chip Networks," in *Proc. IEEE/ACM NoCS*, 2010.

[20] D. A. Miller, "Device Requirements for Optical Interconnects to Silicon Chips," *Proceedings of the IEEE*, vol. 97, no. 7, pp. 1166–1185, 2009.

[21] G. Mora, J. Flich, J. Duato, P. Lopez, E. Baydal, and O. Lysne, "Towards an Efficient Switch Architecture for High-Radix Switches," in *Proc. ACM/IEEE ANCS*, 2006.

[22] T. Moscibroda and O. Mutlu, "A Case for Bufferless Routing in On-Chip Networks," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 196–207.

[23] E. Oki, Z. Jing, R. Rojas-Cessa, and H. J. Chao, "Concurrent Round-Robin-Based Dispatching Schemes For Clos-Network Switches," *IEEE/ACM Transactions on Networking*, vol. 10, no. 6, 2002.

[24] P. Pappu, J. S. Turner, and K. Wong, "Work-Conserving Distributed Schedulers for Terabit Routers," in *Proc. ACM SIGCOMM*, Portland, Oregon, Sep. 2004.

[25] G. Passas and M. Katevenis, "Asynchronous Operation of Bufferless Crossbars," in *Proc. IEEE HPSR*, Jun. 2007.

[26] G. Passas, M. Katevenis, and D. Pnevmatikatos, "Crossbar NoCs are Scalable Beyond 100 Nodes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 4, pp. 573–585, 2012.

[27] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini, "Bringing NoCs to 65 nm," *IEEE Micro*, vol. 27, no. 5, pp. 75–85, 2007.

[28] R. Rajamony, L. B. Arimilli, and K. Gildea, "PERCS: The IBM POWER7-IH High-Performance Computing System," *IBM Journal of Research and Development*, vol. 55, no. 3, pp. 3–1, 2011.

[29] R. Rojas-Cessa, E. Oki, and H. J. Chao, "On the Combined Input-crosspoint Buffered Switch with Round-Robin Arbitration," *IEEE Transactions on Communications*, vol. 53, no. 11, pp. 1945–1951, 2005.

[30] S. Scott, D. Abts, J. Kim, and W. J. Dally, "The Blackwidow High-Radix Clos Network," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, 2006, pp. 16–28.

[31] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. F. Wenisch, D. Sylvester *et al.*, "Swizzle-Switch Networks for Many-Core Systems," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 2, no. 2, pp. 278–294, 2012.

[32] Y. Tamir and G. L. Frazier, "Dynamically-Allocated Multi-Queue Buffers for VLSI Communication Switches," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 725–737, 1992.

[33] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks," in *Proc. ACM SIGCOMM*, Helsinky, Finland, August 2012.